Mapping collections and entity associations

- **Two important (and sometimes difficult to understand) topics didn't appear in the previous lectures:**
  - *the mapping of collections, and*
  - *the mapping of associations between entity classes.*

- Most developers new to Hibernate are dealing with collections and entity associations for the first time when they try to map a typical *parent/child relationship.*

- But instead of jumping right into the middle, we start this lecture with basic collection mapping concepts and simple examples.

- After that, you'll be prepared for the first collection in an entity association

ValueType vs. EntityType

- An object of *value type has no database identity;*
  - *it belongs to an entity instance*
  - its persistent state is embedded in the table row of the owning entity.
- *If an entity class has a collection of value types (or a collection of references to value-typed instances), you need an additional table, the so-called collection table.*
- Before you map collections of value types to collection tables, remember that value-typed classes don't have identifiers or identifier properties.
- The lifespan of a value-type instance is bounded by the lifespan of the owning entity instance.
  - A value type doesn't support shared references.

- **Java has a rich collection API, so you can choose the collection interface and implementation that best fits your domain model design.**

- A **"value type"** is something that doesn't have it's own independent life cycle, and is unique to the object to which it is linked to.
- If the object is destroyed, then so is the value type connected with it.
- As a result, value types are not shared between different objects.

- An **"entity type"** is different.
- It has its own id in the database, and so it's own lifecycle, and can be shared.

- For example, If two users live in the same apartment, they each have a reference to their own homeAddress instance.
- The most obvious value types are classes like Strings and Integers.

- Lets assume there is a user object, which contains an address String. This is at the application level.
  - *1) you implement it as a value type.*
  - *2) you implement is as an entity type.*

1) you implement it as a value type.

- If the user's object is deleted, then so is their address

- It cannot live without the user, nor can it be shared.

- Typically, this would be implemented as simply a column in the same table.

- You would have the name, age, address.

- In the same way that the age is meaningless if the row is deleted, the address is the same.

2) you implement is as an entity type.

- This would likely be implemented in a separate table
  - with the user's row containing a FK to the PK of the address table containing the addresses.

- This would allow
  - the addresses to still exist in their own right without the user and
  - two users to share the same address.

Let's walk through the most common collection mappings.

- **Suppose that Users in a web site are able to attach images to Items.**

- An image is accessible only via the containing item
  - it doesn't need to support associations from any other entity in your system.

- The application manages the collection of images through the Item class, adding and removing elements.

- An image object has no life outside of the collection
  - it's dependent on an Item entity

- **In this case, it isn't unreasonable to model the image class as a value type.**

- **Next. you need to decide what collection to use.**

Out of the box, Hibernate supports the most important JDK collection interfaces.

- In other words, it knows how to preserve the semantics of JDK collections, maps, and arrays in a persistent fashion.

- Each interface has a matching implementation supported by Hibernate, and it's important that you use the right combination.

- Hibernate only *wraps the collection object you've already initialized on declaration* of the field (or sometimes replaces it, if it's not the right one).

- Without extending Hibernate, you can choose from the following collections:
  - java.util.Set
  - java.util.SortedSet
  - java.util.List
  - java.util.Collection
  - java.util.Map
  - java.util.SortedMap

JDK collection interfaces.

- **java.util.Set is mapped with a \<set\> element.**
  The order of its elements isn't preserved, and duplicate elements aren't allowed.
  This is the most common persistent collection in a typical Hibernate application.
  Initialize the collection with a java.util.HashSet.

- **java.util.SortedSet can be mapped with \<set\>**
  The sort attribute can be set to either a comparator or natural ordering for in-memory sorting.
  Initialize the collection with a java.util.TreeSet instance.

- **java.util.List can be mapped with \<list\>**
  Preserves the position of each element with an additional index column in the collection table.
  Initialize with a java.util.ArrayList.

- **java.util.Collection can be mapped with \<bag\> or \<idbag\>.**
  Java doesn't have a Bag interface or an implementation;
  however, *Collection* allows bag semantics (possible duplicates, no element order is preserved).
  Hibernate supports persistent bags (it uses lists internally but ignores the index of the elements).
  Use a java.util.ArrayList to initialize a bag collection.

- **java.util.Map can be mapped with \<map\>**
  Preserves key and value pairs.
  Use a java.util.HashMap to initialize a property.

- **java.util.SortedMap can be mapped with \<map\>**
  The sort attribute can be set to either a comparator or natural ordering for in-memory sorting.
  Initialize the collection with a java.util.TreeMap instance.

Arrays

- **Arrays are supported by Hibernate with**
  - **<primitive-array> (for Java primitive value types) and**
  - **<array> (for everything else).**
- However, they're rarely used in domain models
  - because Hibernate can't wrap array properties.
- You lose lazy loading without bytecode instrumentation, and optimized dirty checking, essential convenience and performance features for persistent collections.
- If you want to map collection interfaces and implementations not directly supported by Hibernate, you need to tell Hibernate about the semantics of your custom collections.
- The extension point in Hibernate is called Persistent-Collection;
  - usually you extend one of the existing PersistentSet, Persistent-Bag, or PersistentList classes.
- Custom persistent collections are not very easy to write and not recommended doing this if you aren't an experienced Hibernate user.

I - *Mapping a set*

- The simplest implementation is a <mark>Set of String image filenames.</mark>
- First, add a collection property to the Item class:

```
private Set images = new HashSet();

. . .

. . .

public Set getImages()

{

  return this.images;

}

public void setImages(Set images)

{

  this.images = images;

}
```

Mapping a set

- Now, create the following mapping in the Item's XML metadata:

```xml
<set name="images" table="ITEM_IMAGE">
   <key column="ITEM_ID"/>
   <element type="string" column="FILENAME" not-null="true"/>
</set>
```

- The image filenames are stored in a table named ITEM_IMAGE, the collection table.
- From the point of view of the database, this table is a separate entity, a separate table, but Hibernate hides this for you.
- The <key> element declares the foreign key column in the collection table that references the primary key ITEM_ID of the owning entity.
- The <element> tag declares this collection as a collection of value type instances—in this case, of strings.

A set can't contain duplicate elements, so the primary key of the `ITEM_IMAGE` collection table is a composite of both columns in the `<set>` declaration: `ITEM_ID` and `FILENAME`. You can see the schema in figure 6.1.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	FILENAME
1	fooimage1.jpg
1	fooimage2.jpg
2	barimage1.jpg

Figure 6.1
Table structure and example data for a collection of strings

It doesn't seem likely that you would allow the user to attach the same image more than once, but let's suppose you did. What kind of mapping would be appropriate in that case?

2, *Mapping an identifier bag*

- An unordered collection that permits duplicate elements is called a *bag.*
- *Curiously,* the Java Collections framework doesn't include a bag implementation.
- However, the java.util.Collection interface has bag semantics, so you only need a matching implementation. You have two choices:

1- Write the collection property with the java.util.Collection interface, and initialize it with an ArrayList of the JDK.

- Map the collection in Hibernate with a standard <bag> or <idbag> element.
- Hibernate has a built-in PersistentBag that can deal with lists; however, consistent with the contract of a bag, it ignores the position of elements in the ArrayList.
- In other words, you get a persistent Collection.

2- Write the collection property with the java.util.List interface, and initialize it with an ArrayList of the JDK.

- Map it like the previous option, but expose a different collection interface in the domain model class.
- This approach works but isn't recommended, because clients using this collection property may think the order of elements is always preserved, which isn't the case if it's mapped as a <bag> or <idbag>.

We recommend the first option. Change the type of images in the Item class from Set to Collection, and initialize it with an ArrayList:

```
private Collection images = new ArrayList();
...
public Collection getImages() {
    return this.images;
}

public void setImages(Collection images) {
    this.images = images;
}
```

Note that the setter method accepts a Collection, which can be anything in the JDK collection interface hierarchy. However, Hibernate is smart enough to replace this when persisting the collection. (It also relies on an ArrayList internally, like you did in the declaration of the field.)

You also have to modify the collection table to permit duplicate FILENAMEs; the table needs a different primary key. An <idbag> mapping adds a surrogate key column to the collection table, much like the synthetic identifiers you use for entity classes:

```
<idbag name="images" table="ITEM_IMAGE">

    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>

    <key column="ITEM_ID"/>

    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

ITEM	
ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE		
ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	1	fooimage1.jpg
2	1	fooimage1.jpg
3	3	barimage1.jpg

Figure 6.2   A surrogate primary key allows duplicate bag elements.

In this case, the primary key is the generated `ITEM_IMAGE_ID`, as you can see in figure 6.2. Note that the `native` generator for primary keys isn't supported for `<idbag>` mappings; you have to name a concrete strategy. This usually isn't a problem, because real-world applications often use a customized identifier generator anyway. You can also isolate your identifier generation strategy with placeholders; see chapter 3, section 3.3.4.3, "Using placeholders."

Also note that the `ITEM_IMAGE_ID` column isn't exposed to the application in any way. Hibernate manages it internally.

A more likely scenario is one in which you wish to preserve the order in which images are attached to the `Item`. There are a number of good ways to do this; one way is to use a real list, instead of a bag.

Mapping a list

First, let's update the Item class:

```java
private List images = new ArrayList();

...

public List getImages() {
  return this.images;
}

public void setImages(List images) {
  this.images = images;
}
```

Mapping a list

- A <list> mapping requires the addition of an *index column to the collection table.*

- The index column defines the position of the element in the collection.

- Thus, Hibernate is able to preserve the ordering of the collection elements.

Map the collection as a <list>:

```
<list name="images" table="ITEM_IMAGE">
    <key column="ITEM_ID"/>
    <list-index column="POSITION"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

(There is also an index element in the XML DTD, for compatibility with Hibernate 2.x. The new list-index is recommended; it's less confusing and does the same thing.)

The primary key of the collection table is a composite of `ITEM_ID` and `POSI-TION`. Notice that duplicate elements (`FILENAME`) are now allowed, which is consistent with the semantics of a list, see figure 6.3.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	POSITION	FILENAME
1	0	fooimage1.jpg
1	1	fooimage2.jpg
1	2	foomage3.jpg

Figure 6.3   The collection table preserves the position of each element.

The index of the persistent list starts at zero. You could change this, for example, with `<list-index base="1".../>` in your mapping. Note that Hibernate adds null elements to your Java list if the index numbers in the database aren't continuous.

Alternatively, you could map a Java array instead of a list. Hibernate supports this; an array mapping is virtually identical to the previous example, except with different element and attribute names (`<array>` and `<array-index>`). However, for reasons explained earlier, Hibernate applications rarely use arrays.

4,

Now, suppose that the images for an item have user-supplied names in addition to the filename.

One way to model this in Java is a map, with names as keys and filenames as values of the map.

Mapping a map

Again, make a small change to the Java class:

```java
private Map images = new HashMap();
...
public Map getImages() {
    return this.images;
}

public void setImages(Map images) {
    this.images = images;
}
```

Mapping a <map> is similar to mapping a list.

ITEM

ITEM_ID	NAME
1	Foo
2	Bar
3	Baz

ITEM_IMAGE

ITEM_ID	IMAGENAME	FILENAME
1	Image One	fooimage1.jpg
1	Image Two	fooimage2.jpg
1	Image Three	foomage3.jpg

Figure 6.4   Tables for a map, using strings as indexes and elements

```xml
<map name="images" table="ITEM_IMAGE">

    <key column="ITEM_ID"/>

    <map-key column="IMAGENAME" type="string"/>

    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The primary key of the collection table is a composite of ITEM_ID and IMAGENAME. The IMAGENAME column holds the keys of the map. Again, duplicate elements are allowed; see figure 6.4 for a graphical view of the tables.

This map is unordered.

What if you want to always sort your map by the name of the image?

5. *Sorted and ordered collections*

□ In a startling abuse of the English language, the words *sorted and ordered mean different* things when it comes to Hibernate persistent collections.

□ A *sorted collection is* sorted in memory using a Java comparator.

□ An *ordered collection is ordered at the* database level using an SQL query with an order by clause.

SortedMap

Let's make the map of images a sorted map.

First, you need to change the initialization of the Java property to a java.util.TreeMap and switch to the java.util.SortedMap interface:

```
private SortedMap images = new TreeMap();
...
public SortedMap getImages() {
return this.images;
}
public void setImages(SortedMap images) {
this.images = images;
}
```

SortedMap

this collection accordingly, if you map it as sorted:

```xml
<map name="images" table="ITEM_IMAGE" sort="natural">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

By specifying sort="natural", you tell Hibernate to use a SortedMap and to sort the image names according to the compareTo() method of java.lang.String.

If you need some other sort algorithm (for example, reverse alphabetical order), you may specify the name of a class that implements java.util.Comparator in the sort attribute.

For example:

```
<map name="images" table="ITEM_IMAGE" sort="auction.util.comparator.ReverseStringComparator">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

A java.util.SortedSet (with a java.util.TreeSet implementation) is mapped like this:

```
<set name="images" table="ITEM_IMAGE" sort="natural">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Alternatively, instead of switching to the Sorted* interfaces (and the Tree* implementations), you may want to work with a linked map and to sort elements on the database side, not in memory.

Keep the Map/HashMap declaration in the Java class, and create the following mapping:

```
<map name="images" table="ITEM_IMAGE" order-by="IMAGENAME asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

The expression in the order-by attribute is a fragment of an SQL order by clause.

In this case, Hibernate orders the collection elements by the IMAGENAME column in ascending order during loading of the collection.

You can even include an SQL function call in the order-by attribute:

```
<map name="images" table="ITEM_IMAGE" order-by="lower(FILENAME) asc">
    <key column="ITEM_ID"/>
    <map-key column="IMAGENAME" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

□ You can order by any column of the collection table.

    □ Internally, Hibernate uses a LinkedHashMap, a variation of a map that preserves the insertion order of key elements.

    □ In other words, the order that Hibernate uses to add the elements to the collection, during loading of the collection, is the iteration order you see in your application.

□ The same can be done with a set:

    □ Hibernate internally uses a LinkedHashSet. In your Java class, the property is a regular Set/HashSet, but Hibernate's internal wrapping with a LinkedHashSet is again enabled with the order-by attribute:

```
<set name="images" table="ITEM_IMAGE" order-by="FILENAME asc">
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

- You can also let Hibernate order the elements of a bag for you during collection loading.

- Your Java collection property is either Collection/ArrayList or List/ ArrayList.

- Internally, Hibernate uses an ArrayList to implement a bag that preserves insertion-iteration order:

```xml
<idbag name="images" table="ITEM_IMAGE" order-by="ITEM_IMAGE_ID desc">
    <collection-id type="long" column="ITEM_IMAGE_ID">
        <generator class="sequence"/>
    </collection-id>
    <key column="ITEM_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</idbag>
```

- The linked collections Hibernate uses internally for sets and maps are available only in JDK 1.4 or later; older JDKs don't come with a LinkedHashMap and LinkedHashSet.

- Ordered bags are available in all JDK versions; internally, an ArrayList is used.

- In a real system, it's likely that you'll need to keep more than just the image name and filename.

- You'll probably need to create an Image class for this extra information.

- This is the perfect use case for a collection of components.